Python Introduction

or
Stuffing Your Brains Full
with
Things You will Use in the Lab Course

Kent Engström
kent@unit.liu.se

What Do I Expect From You?

- I assume that you:
  - know how to program in at least one other program language
  - want to learn Python
  - will practice Python programming soon
  - can absorb a lot of stuff, forget about the details, but remember that you can look them up
    - in the book
    - on-line

Why Python?

- It's easy to learn and use
- It's fast to develop in
- It's readable (even for human beings)
- It's portable and free (open source)
- It has a good mix of working features from several program language paradigms
- “Batteries Included”

Why Python? (contd)

- It's easy to use Python to control other software ("scripting language")
- It's easy to embed Python in other programs or vice versa
- It's fun to program in Python

Any Drawbacks?

- Python is not a run-time speed daemon compared to e.g. C or Fortran.
  - Does it matter for your application?
  - You can mix Python and other languages to get the best of both worlds
- The dynamic typing is not trusted by static typing enthusiasts
- Some newbies do not like the indentation-based syntax

Relatives

- Python is mostly a procedural language, like C and Fortran.
- Python is also object-oriented like Java and C++ (but you can ignore OO if you don't need it)
- Python has some functional language features (higher-order functions etc)
- Python allows you to get work done (like Perl) by giving you access to the operating system, Internet protocols etc.

Python is Byte-Compiled

- Python is a byte-compiled language
  - The source is transformed to instructions for a fictional Python-optimized CPU
  - Like Java, but not as focused on the byte-code as a portable program delivery mechanism
  - The source code is compiled when it is run, so you will not have to invoke a compiler first
  - The byte-code is saved for reuse (if you do not change the source between runs)

Interactive Use

- You can start a Python interpreter and start typing statements and expressions into it and see the answers directly:

```
% python2.3
Python 2.3.4 (#1, Oct 26 2004, 16:42:40)
[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> x=42.17
>>> x*x+10
1788.3089000000002
>>>
```

Python Scripts

- Python scripts can be made executable in the same way as shell scripts:
  - Make the file executable:
    - `chmod +x myprog.py`
  - Add a suitable interpreter line at the top of the file. For portability reasons, use:
    - `#!/usr/bin/env python`

Getting Help

- Use the on-line docs
  - http://www.python.org/doc
- Read the book you have
- Use the help() function interactively
  - help(“open”) # help about the “open” function

A Small Example

```
seen = {}
num_dup = 0
f = open("/etc/passwd")
for line in f:
    (user, rest) = line.split(":",1)
    if user in seen:
        print "User",user,"has been seen before!"
        num_dup += 1
    else:
        seen[user]= 1 # Any value would do
print "Number of duplications:", num_dup
```

Python Types

Python Types

- We will take a look at the basic data types (and operations) available in Python before we dig into the syntax.
- Python uses *dynamic typing*:
  - Variables refer to objects.
  - Variables as such has no type.
  - Objects have types (integers, strings, lists etc). You cannot add 1 and “3”.

Integers

- Python has two kinds of integers:
  - *int* (the normal kind, like C)
    - Most likely 32 or 64 bits, signed
  - *long* (like LISP bignum, bounded by memory size only)
    - Written and displayed as an integer with an “L” after
- In modern Python, an *int* is automatically converted to a *long* if it would overflow

Integer Literals

- 4711 (an int)
- 4711L (a long entered explicitly)
- 0x1F (hexadecimal, decimal 31)
- 010 (octal, decimal 8)
  - Do not add leading zeros if you do not intend the integer to be treated as octal!

Floating Point Numbers

- The Python *float* format is equivalent to the C *double* on the platform
- Literals written as you would expect (including "E notation").
- Python does not hide binary floating point quirks:
  - If you enter 0.1 at the interactive prompt, it may be echoed as 0.10000000000000001
  - You should know why better than me ! :-)

Complex Numbers

- Python has a *complex* type (formed from two C doubles).
- The imaginary part is entered with a “j” appended:
  - `z=3.1+5.6j`
- Parts can be accessed like this:
  - `z.real, z.imag`

Operations

- The normal stuff: +, -, *, /, % (modulo), ** (power)
- Beware: a division between two integers is carried out as an integer division:
  - 8/3 => 2
  - 8.0/3 or 8/3.0 or 8.0/3.0 => 2.6666666666666665
  - This is going to change in future Python versions:
    - 8/3 => 2.6666666666666665
    - 8//3 => 2

Operations (contd)

- Bitwise stuff: &, |, ^, ~, <<, >>
- Absolute value: abs(x)
- Conversions: int(x), long(x), float(x), complex (re,im)
- Rounding: round(1.2345,2) gives 1.23

The math and cmath Modules

- More functions are present in the *math* module, that need to be imported ("import math") first.
- Example:
  - math.sin(x), math.exp(x), math.sqrt(x)
- Complex versions are present in the *cmath* module
  - math.sqrt(-1) raises an exception
  - cmath.sqrt(-1) gives 1j

Strings

- Strings are a central data type in Python (as well as in all similar languages)
- Strings store 8-bit characters or bytes
  - Null (ASCII 0) bytes are not special like in C
  - Long strings are allowed
- Strings are *immutable*, i.e. they cannot be changed after they have been created
  - We can create new strings from parts of existing ones, of course

No Character Type

- Python has no character data type
  - Strings are sequences of “characters”
  - “Characters” are strings of length 1
- Works perfectly OK in practical life
- Conversion between “characters” and integers:
  - ord('A') gives 65
  - chr(65) gives 'A'

String Literals

- 'Single quotes around a string'
- ”Double quotes around a string also OK”
- 'This is not OK”, ”Nor is this'
- 'Can have the ”other type” inside.'
- ”Like 'this' too.”
- '''Triple single quotes allows newline inside string (not removed)'''
- ”””Triple double quotes also OK.”””

Escapes and Raw Strings

- Backslashes for special characters, mostly like in C:
  - 'Line 1\nLine 2 after a newline'
  - 'This \\ will become a single backslash'
  - An unknown character after a backslash is not removed
- In raw strings backslashes are not special (good for regular expressions and Windows file paths):
  - r'This single \ gives a backslash here'
  - r'\n' (length two string with a backslash and an 'n')

Basic String Operations

- len(s) gives the length of the string
- s+t concatenates two strings
- s*n repeats a string n times
  - '='*10 gives '=========='
- str(x) converts x (e.g. a number) to a string
- s in t gives true iff s is a substring of t

Formating

- Like printf in C: format % args
  - 'f(%f) = %f' % (x, value)
  - 'Integer %d and a string: %s' % (n, s)
- The thing on the right side is a *tuple* (we will return to them later)

Indexing

- To get a “character” from a string, we use (zero-based) indexing:
  - s[0] is the first (leftmost) character in the string
  - s[1] is the second character in the string
- Negative indexes work from the end:
  - s[-1] is the last character in the string
  - s[-2] is the penultimate character in the string
- Indexing outside of the string raises an *IndexError* exception

Slicing

- Substrings are extracted using slicing:
  - 'Python'[1:4] gives 'yth' (not 'ytho'!)
  - Imagine that the indices are between characters
- Omitted indices default to beginning or end:
  - 'Python'[3:] gives 'hon'
  - 'Python'[:3] gives 'Pyt'
  - 'Python'[:] gives 'Python'

Slicing (contd)

- Negative indices work here too:
  - 'Python'[1:-1] gives 'ytho':
- Slicing outside of the strings does not raise exceptions:
  - 'Python'[4:10] gives 'on'
  - 'Python'[8:10] gives '' (an empty string)

String Methods

- String objects have methods we can call (object-oriented!).
- s.upper() converts s to upper case (returning a new string)
  - 'Python'.upper() gives 'PYTHON'
- s.lower(), s.capitalize() also available
- s.find(t) gives index of first substring t in s:
  - 'Python'.find('th') gives 2
  - 'Python'.find('perl) gives -1 (meaning: not found)

String Methods (contd)

- s.replace(from,to) replaces all occurences:
  - 'ABCA'.replace('A','DE') gives 'DEBCDE'
- s.strip() removes whitespace from beginning and end
- s.lstrip() and l.rstrip() only strips from beginning and end respectively

Splitting and Joining

- s.split() splits on whitespace
  - 'Python rules'.split() gives ['Python', 'rules']
  - The result is a *list* of strings
- s.split(sep) splits on the specified separator string
  - a_long_string.split('\n') splits the string into lines
- sep.join(list) joins a list of strings using a separator:
  - ':'.join(['1','B','3']) gives '1:B:3'

Unicode Strings

- A separate type *unicode* is used to hold Unicode strings
  - u'ASCII literals OK'
- Conversion examples:
  - u.encode('latin-1') converts to plain string in Latin-1 encoding
  - u.encode('utf-8') converts to UTF-8 coding
  - s.decode('utf-8') converts the plain strings from UTF-8 to a unicode string

Lists

- Lists are ordered collections of arbitrary objects
- Lists are not immutable, thus they can be changed in-place
- Like vectors (one dimensional arrays) in other languages
- Not linked lists like in LISP (accessing the last element is not more expensive than accessing the first)

Lists (contd)

- Lists have a certain size but can grow and shrink as needed
  - No holes, though: You cannot add a fourth element to a two element list without first adding a third element.
- Adding or removing elements at the end is cheap
- Adding or removing elements at the beginning is expensive

Indexing and Slicing

- Indexing and slicing works like for strings:
  - mylist = ['Zero', 1, 'Two', '3']
  - mylist[2] gives 'Two' (the element)
  - mylist[2:] gives ['Two','3'] (a new list)
- We can change a list in-place using indexing to the left of =:
  - mylist[2] = 'Zwei'
  - mylist is now ['Zero', 1, 'Zwei', '3']
- Slices also work!

List Operations

- Many string operations work here too:
  - len(mylist) gives the length
  - l1+l2 concatenates two lists
  - mylist*n repeats the list
    - [1,2] * 3 is [1,2,1,2,1,2]
- Other string operations are not available:
  - No list.upper() etc

Adding Elements

- mylist.append(elem) appends an element at the end of the list
  - [1,2,3].append(4) gives [1,2,3,4]
- mylist.extend(otherlist) appends a whole list at the end
  - [1,2,3].extend([4,5]) gives [1,2,3,4,5]
- mylist.insert(pos, elem) inserts an element at a certain position
  - [1,2,3].insert(0, 'zero') gives ['zero',1,2,3]

Deleting Elements

- mylist.pop() deletes the last element in-place and returns it:
  - mylist = [1,2,3,4]
  - mylist.pop() gives 4 and mylist is now [1,2,3]
- mylist.pop(n) deletes element n:
  - mylist.pop(1) gives 2 and mylist is now [1,3]

Reversing and Sorting

- mylist.reverse() and mylist.sort() reverses and sorts lists in-place, i.e. they return no value but they change the list
  - mylist = [2,1,3,4]
  - mylist.sort() gives no return value, but mylist is now [1,2,3,4]
  - mylist.reverse() gives no return value, but mylist is now [4,3,2,1]

Tuples

- Tuples are like lists, but they are immutable (like strings)
- Literals are written using comma with parenthesis as needed:
  - () is the empty tuple (parenthesis needed!)
  - (1,) is a tuple containing a single element (note trailing comma)
  - (1,2) is a tuple containing two elements
  - (1,2,) is the same thing (trailing comma allowed but not needed)

Tuple Operations

- len(t), t1+t2, t*n works
- indexing and slicing works (for access but not for changing)
- No methods available

Lists vs Tuples

- Use lists for dynamic sequences of “similar” things, i.e. a list of students attending a course.
- Use tuples for fixed size sequences of “different” things, i.e.
  - a tuple of coordinates in 3D space,
  - a tuple of student name and student test score

Nesting

- Lists and tuples (and other things) can be arbitrarily nested:
  - x=[1,['foo',2],(3,[4,5])]
  - x is a list of an integer, a list and a tuple
  - x[1] is a list of a string and an integer
  - x[2] is a tuple of an integer and a list
  - x[2][1] is a list of two integers

Dictionaries

- Dictionaries (type *dict*) are associative arrays
  - Perl programmers call their version *hashes*
- A dictionary can be indexed by any immutable type, not just integers
- Literals:
  - d1={} stores the empty dictionary in d1
  - d2={1:2, 'foo': 3}
  - d2 now maps the integer 1 to the integer 2, and the string 'foo' to the integer 3

Indexing

- d2['foo'] gives 3
- d2['bar'] raises *KeyError*
- d2[1] = 10 overwrites the value for key 1
- d2[2] = 20 adds a new value for the key 2 (not present before)
- del d2[1] deletes the item for key 1
- Slicing does not work as there is no concept of order between the items in a dictionary

Avoiding KeyError

- key in dict return true iff an item for the key is present in the dictionary
- dict.get(key) works like dict[key] but returns None (a special null object) if no item for key is present
- dict.get(key,default) returns the specified default value instead of None if the item is not present

Getting Keys, Values or Items

- We can get the keys, values or items from a dictionary (the order is not guaranteed absolutely but consistent between the methods):
  - d={1:2,10:20}
  - d.keys() is [1,10]
  - d.values() is [2,20]
  - d.items() is [(1,2),(10,20)]

Overview of Container Types

- Sequences
  - Immutable sequences
    - Strings
      - *str*: plain strings
      - *unicode*: Unicode strings
    - *tuple*: tuples
  - Mutable sequences
    - *list*: lists
- Mappings
  - *dict*: dictionaries

None

- None is the only value of the type *NoneType*.
- It is used in multiple places to mean N/A, data missing, do not care, etc.
- If a function does not return a value, it returns None implicitly.
- A variable containing None is not the same thing as a variable not being defined at all

Other Types

- We will encounter the *file* type later
- Internal types for things like
  - functions
  - modules
  - classes, instances and methods
  - even more internal stuff
- Types defined by extension modules
  - e.g. images, database connections

Python Statements

Statements

- Python programs consists of statements, e.g.
  - assignments like x=10
  - print statements to output things
  - if statements for selection
  - while or for for loops
- Statements have no values (we cannot speak of the value of a print statement or an assignment)
- Statements have “side effects”

Expressions

- Statements can contain expressions (things that have a value):
  - n=n+1 (where n+1 is an expression used to calculate the value we are to assign to n)
  - print math.sin(x*10)

Expression Statements

- An expression can be used as a statement in a program
  - n+1 is a valid statement but utterly useless in a program (calculate n+1 and throw the value away)
- This is mostly used to call functions (a function call is an expression):
  - process_file('myfile.txt')
  - If the function happened to return a value, we threw it away above

No “Statement Expressions”

- We cannot have statements (e.g. assignments) inside expressions in Python.
- This means that we cannot use the following trick from C:
  - if ((var=getsome() == 0) ...
- This protects us from common errors like this:
  - if var=1

Some Basic Syntax

- Comments begin at a # characters and continues to the end of the line
- No semicolon needed at the end of the line
  - But we can use it to string together statements on the same line:
    - a=10; b=20; c=(a+b)/2
- Backslash at end of line allows us to continue a line
  - This is not needed inside a "parenthetical expression" started by (, [or {.

Assignment Statements

- The basic form is written as var=expression, e.g.
  - x=10
  - n=n+1
  - s=s+'\n' + s2.strip() + ':'
- Assignment uses =, equality testing uses ==
- Variable names
  - begin with a character or underscore
  - continues with characters, digits and/or underscores
  - are case sensitive

“Fancy” Assignments

- Multiple assignments work:
  - x=y=z=0
- Decomposing lists and tuples work:
  - t=(1,2)
  - x,y=t means x=1, y=2
- We can use this to swap to variables:
  - x,y = y,x

Augmented Assignments

- x += 1 works like x=x+1
- x *= 2 works like x=x*2
- But: mutable objects *may* be changed in-place
  - list += [4,5] behaves like list.extend([4,5]) not list=list+[4,5]
- There is no n++ or ++n like in C.

Values and References

- A variable contains a reference to an object, not the value as such
- This is boring as long as we use only immutable objects:
  - a=1 # create an object with value 1, store reference in a
  - a=a+2 # get object refered to by a, get object with value 2, perform addition to get a new object with value 3, store reference to that object in variable a

Aliasing

- But what can happen when the objects are mutable?
  - a=[1,2,3] # create a list, store a reference to it in a
  - b=a # store the same reference in variable b
  - b[0]=10 # get the list referenced by b, change element 0...
  - a[0] is of course also 10 now, as a and b refers to the same list object!

Aliasing (contd)

- Often, this is what we want, but sometimes we need to copy a mutable object so we do not change the original when doing operations on the copy. Use
  - mylist[:] to get a copy of the list mylist
  - mydict.copy() to get a copy of the dictionary mydict
- These are *shallow* copies
- New Python programmers tend to be too concerned about copies and aliasing

Garbage Collection

- We never need to deallocate objects explicitly.
- When the last reference to an object goes away, it is deleted and its memory reclaimed:
  - s=”Waste”*10000 # create a big string
  - t=(1,s) # a reference to s is in the tuple now
  - s=1 # we lost one reference to the big string but the one in the tuple remains
  - t=(1,2) # we now lost the last reference to the string and it is deleted.

Print Statements

- A simple way to output data to the *standard output* is provided by the print statement:
  - print 10
  - print x
  - print 'Value of', varname, 'is', value
  - print 'Value of %s is %s' % (varname, value)
  - print 'Newline at end of this'
  - print 'No newline at end of this',

Conditional Statements

- Python provides an if-elif-else-statement:

```
# Plain if statement
if temp < 10:
    print “Temperature too low.”

# Dual if-else statement
if x < 0:
    print “No roots can be found.”
else:
    print “Will solve for roots.”
```

Conditional Statements (contd)

```
# Multiple choices
if temp < 10:
    print “Temperature too low.”
    start_heater()
elif temp < 30:
    print “Temperature OK.”
elif temp < 100:
    print “Temperature too high.”
    start_air_conditioner()
else:
    print “We are boiling!”
    evacuate_building()
```

Indentation Sensitive Syntax

- You saw no braces or begin-end pairs delimiting the statements in the compound if-elif-else statement
- Python uses the indentation itself to infer program structure.
- This is smart, as you should always indent your code properly!
- The Python mode in Emacs supports this, so it is no big deal if you use the One True Editor.

Nested Compound Statements

- This is what a nested compound statement looks like.

```
if a == b:
    print "A and B are equal."
    if b == c:
        print "All three are equal!"
    else:
        print "But C is different!"
elif a < b:
    print "A is smaller than B."
else:
    print "A is greater than B."
```

Comparison Operators

- We have the usual set of operators to compare things:
  - == tests for equality
  - != (or <>) tests for inequality
  - <, <=, >, >= are also there
  - Numbers are compared without caring about type: 0 == 0.0, 0.0 == 0j
  - Sequences are compared lexicographically: (1,2) < (2,1)

Booleans

- The comparison operators return values of type *bool*: True or False.
- Earlier versions of Python used 1 for True and 0 for False.
- Compatibility Hack: *bool* is a sublass of *int*, where 1 is printed as True and 0 as False.
  - True + 10 gives 11, but please do not ever write code like that!

Truth Values

- Python considers every value to be true or false, not only the *bool*s:
  - True is true and False is false, of course
  - Numerical values are false if zero, true otherwise
  - Containers are false if they are zero, true if they contain items.
  - None is false
  - User-defined classes can contain code to determine if they are true or false

Logical Operators

- Python has “and” and “or” operators, short-circuiting like in C:
  - if x>0 and 1/x > 10: ...
  - We do not risk dividing by zero in the second part above. If x is zero, the second part is not evaluated.
- The “not” operator return True when given a false value and False when given a true value:
  - not False gives True
  - not True gives False
  - not 2 gives False (because 2 is a true value)

Pre-tested Loop Statements

- A pre-tested loop where we loop as long as the condition is true (no loops at all if the condition is false the first time around):

```
x=1
while x <= 10:
  print “Line number”,x,”of 10.”
  x+=1
```

Break and Continue Statements

- The break statement to exit the innermost loop immediately.
  - We use “while True:” if we need an endless loop (and then we can exit it using break anyway)
- The continue statement skips the rest of the innermost loop body.
- We cannot use this to exit or skip more than the innermost loop.

Iteration Loop Statements

- To loop over sequences, we do not use the while statement and indexing. Instead, we have the for loop:

```
choices = ['Vanilla', 'Chocolate', 'Lemon']
print 'Choose ice-cream'
print '<UL>'
for c in choices:
    print '<LI>' + c
print '</UL>'
```

Iteration

- The for loop works for all containers
  - list and tuples are iterated element by element
  - strings are iterated “character” by “character”.
  - dictionary iteration is over the keys in an undefined order
- User-defined classes can specify their own iteration behaviour

Break or Else...

- For loops (and while loops too) can have an else: part that is only taken on “normal exit” but not when break is used to exit the loop:

```
for e in long_list:
    if is_good(e):
        print “A good element was found, done.”
        break
else:
    print “No good element was found.”
```

range and xrange

- The range expression lets us use for loops to loop over numerical ranges:
  - range(5) gives [0,1,2,3,4] (five items)
  - range(10,15) gives [10,11,12,13,14]
  - for i in range(1,11): print "Line %d of 10" % i
- If the range is large, it is wasteful to construct the whole list in memory. We can use xrange instead of range then.
  - It creates a "fake list" that works just like the one range builds for the purpose of iteration.

Python Functions

Functions

- Every high-level language have some kind of subroutine concept.
- Python has functions
- Python does not have procedures
  - Functions that end without calling the return statement implicitly returns None.
  - If we do not care about the return value from a functions, it is silently discarded

Functions (contd)

- Functions are defined by def:

```
def origin_distance(x,y):
    return math.sqrt(x*x + y*y)

def print_var(name, value):
    print “Value of”, name, “is”, value

def func_with_no_arg():
    return 42
```

Calling Functions

- Functions are called using parentheses:
  - dist = origin_distance(x1,y1)
  - print_var('x', 4711)
  - answer = func_with_no_arg()
- We cannot omit the parentheses in the last example!
  - We would then assigned the function object to answer, not the result of calling the function
  - Functions are first-class objects that can be stored in variables

Arguments

- Keyword arguments and defaults are possible:

```
def f(x, y, verbose=0, indent=4): ...
f(1,2) # Ok, defaults for verbose and indent
f(1,2,1) # Ok, verbose=1, default for indent
f(1,2, indent=8) # Ok, default for verbose
f(verbose=2, y=2, x=1) # Ok, default for indent
f(1) # Error
f(verbose=2, 1, 2) # Error
```

Call by Value

- Python uses call by value
  - def f(x): x = 3
  - y=2; f(y); print y
  - We will get “2” printed. The assignment to x in f does not change the value outside the function body
- But mutable objects can change:
  - def f(x): x[0] = 3
  - y=[1,2]; f(y); print y
  - We will get [3,2] printed

Local Variables

- A variable assigned in a function is local and does not affect a variable with the same name outside the function:
  - def f(x): z=3
  - z=1; f(0); print z
  - We will get "1" printed.

Accessing Global Variables

- We can access global variables inside a function:
  - def f(x): print g
  - g=”Global!”
  - f(0)
  - This will print “Global!” just like we expected

Assigning to Global Variables

- To be allowed to assign to a global variable we have to declare it using a global statement. The code below will print “17” and then “20”.

```
x = 17
def f():
    global x
    print x
    x = 20
f()
print x
```

Python Modules

Modules

- Programs can be divided into several files.
- Each file defines a *module*.
- Each module has its own global namespace (there is no global namespace above all modules).
  - Modules thus provide namespace isolation so two variables or functions with the same name in two different modules doesn't clash.
- Modules enable code reuse
  - Python already provides a lot of built-in modules for us to use.

Import Statements

- To get access to a module, we use the import statement:
  - import foo
- This imports foo.py
  - from the same directory as the running program or
  - from a directory on the python module path
- After the import, we can refer to global variables, functions etc in foo using “foo.” before the name, like this: foo.fak, foo.x

Import into Our Namespace

- Using a special form we can import some names from a module into our own namespace:
  - from foo import fak, x
  - from math import sin, cos, tan, sqrt, exp
- We can also import all names from a module into our own namespace:
  - from foo import *
- A module can control what names are exported when using the "*" import.

Import Runs... Once

- The first time a module is imported during the running of a program, the code in the module runs:
  - Even def statements defining functions are executable code that is run to perform the defining
- If the module is imported again the code is not run again
  - Only the importer's namespace is updated
- Avoid cyclic module dependencies

Packages

- Complicated modules can be subdivided hierarchically.
- Such modules are called *packages* and are outside the scope of this introduction.

Byte-Compiled Code Saved

- We mentioned earlier that Python code is byte-compiled.
- When a module is imported and thus byte-compiled, the compiled code is saved in a file with a .pyc extension:
  - foo.py is compiled to foo.pyc
  - The byte-compiled code is loaded instead of the source code the next time the module is imported (if the source file has not changed)

Python Object Orientation

Object-Orientation

- Python's Object Orientation
  - is not mandatory to use in your programs
  - has inheritance (even multiple)
  - has not overloading (how would that be possible?)
  - makes all methods virtual (redefinable by subclasses)
  - doesn't really protect object variables from “cheaters”

Class Definition

- Classes are defined and objects created from them like this:

```
from math import sqrt
class Coord:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def origin_distance(self):
        return sqrt(self.x**2 + self.y**2)
    def is_at_origin(self):
        return self.origin_distance() == 0

c1 = Coord(10,20) # create and run __init__
print c1.origin_distance()
```

Classes (contd)

- When we call a method on an object, the corresponding method in the class is called, with the object as an implicit first argument that we get into the self argument.
- Also note the difference between self.x (object attribute) and x (local variable from the argument list) in the __init__ method.

Inheritance

- Let us define a subclass
  - The is_at_origin method now comes from the superclass Coord while we implement origin_distance here:

```
class ManhattanCoord(Coord):
    def origin_distance(self):
        return abs(self.x) + abs(self.y)

c2 = ManhattanCoord(5,5)
if c2.is_at_origin(): print "Impossible!"
```

Emulating Built-in Objects

- By defining certain special methods in our classes, our objects can behave like numbers, lists, etc. Examples:
  - __add__(self, other): addition using +
  - __getitem__(self, index): indexing
  - __len__(self): len(object)

More to Learn

- There is more to learn about OO in Python, of course, such as:
  - Multiple inheritance
  - Static and class methods
  - “New-style” OO (unification of classes and types)
- This is beyond the scope of this introduction.

Python Exceptions

Exceptions

- Python handles errors and other exceptional occurrences by raising exceptions.
- If not caught, they will cause the program to be aborted.

```
>>> a=1/0; print “not reached”
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>>
```

Catching Exceptions

- Exceptions are caught by placing the “dangerous” code in a try:-except: compound statement.
  - If dx should be undefined below, we get a NameError instead, which is not caught by the handler.

```
try:
    slope = dy/dx
    vertical = 0
except ZeroDivisionError:
    slope = None
    vertical = 1
```

Catching Exceptions (contd)

```
try:
    res = dangerous_function()
except (KeyError, NameError):
    print "Trouble type A"
    x=a/b
except ZeroDivisionError:
    print "Trouble type B"
except:
    print "Unknown exception caught"
```

- Multiple handlers can be specified
- The division in the first handler is not protected by the second handler
- Avoid the last kind of handler if possible

Defining Our Own Exceptions

- We define our own exceptions by subclassing the built-in Exception class
  - We can then raise it using a raise statement.
  - The pass statement in the first line is a no-op for use where the syntax requires a statements and we have nothing to do.

```
class MyOwnError(Exception): pass
def f(foo):
    if foo > 100: # Too high
        raise MyOwnError
```

Guaranteed Finalization

- Another form of try: can be used to guarantee that a piece of cleanup code is run regardless of how a dangerous piece of code is executed.

```
def f():
    rsrc = alloc_external_expensive_resource()
    try:
        # This code may raise an exception
        res = call_dangerous_code()
    finally:
        dealloc_resource(rsrc)
```

Python's Included Batteries

File Objects

- You get them with open for normal files:
  - f=open('file.txt') # for reading
  - f=open('file.txt', 'r) # same
  - f=open('file.txt', 'w) # for writing
  - f=open('file.txt', 'a) # for appending
  - f=open('file.txt', 'rb) # b for binary mode on Windows
- Some modules give you file-like objects to play with (e.g. urllib)

File Objects (contd)

- Reading
  - f.read() # reads the whole file
  - f.read(10) # reads 10 bytes
  - f.readline() # reads a line including newline
  - for line in f: ... # modern way of reading line by line
- Writing
  - f.write(string)
- Closing
  - f.close()

Module sys

- Misc system stuff:
  - sys.stdin, sys.stdout, sys.stderr: file objects
  - sys.argv: program name + argument list
  - sys.environ: Unix environment as a dictionary
  - sys.path: Python module search path
  - sys.exit(ret): exit the program with a return code
  - ...

Modules math and cmath

- We have already mentioned these
- If it is in the C math library, it is here too.

Module re

- Regular expressions
  - Perl compatible, to a large extent

```
m = re.match(r'([^=]+) *= *(.*)', line)
if m:
    param, value = m.group(1,2)
else:
    print “Bad configuration line found”
```

Module struct

- Handle binary data structures (in files etc)

```
# Pack into 16-bit unsigned, big endian
b = struct.pack(">HH", 640, 480)
# b is '\x02\x80\x01\xe0'

# Unpack them again
(w, h) = struct.unpack(">HH", b)
# w is 640, h is 480
```

Module random

- Pseudorandom numbers:
  - i = random.randrange(10,20) # 10 <= i < 20
  - r = random.random() # 0.0 <= r < 1.0
  - dir = random.choice([“left”, “right”, “up”, “down”])
  - random.shuffle(list)
  - random.seed(something)

Operating System Access

- Basic OS access
  - getcwd, chdir, getpid, getuid, setuid...
  - rename, unlink, ...
  - system, fork, exec, ...
- Path handling in os.path
  - dir, file = os.path.split(path), ...
- More similar modules:
  - time, stat, glob, fnctl, pwd, grp, signal, select, mmap, tty, pty, crypt, resource, nis, syslog, errno, tempfile, ...

Running Commands

- os.system(runthis)
- Module popen2
- Module commands
- Module subprocess in Python 2.4

Threading

- thread
  - Low level thread support
- threading
  - Higher level (more like Java)
  - Synchronization primitives
- Queue
  - Thread-safe queue

Internet Protocols and Format

- socket
- urllib, urllib2
- httplib, ftplib, gopherlib
- cgi, Cookie
- poplib, imaplib, smtplib, nntplib, telnetlib, dnslib
- email, mimetools, mailbox, mhlib
- binhex, uu, binascii, base64, quopri
- xdrlib, gzip, zlib

Even More Stuff Included

- Serialising
- XML support
- Testing
- Profiling
- TkInter GUI
- Option parsing
- …

Available on the Net

- Database Glue (MySQL, PostgreSQL, ...)
- Python Imaging Library (PIL)
- Numarray: array handling and computations
- ...

Thanks for Not Falling Asleep

Kent Engström
kent@unit.liu.se